AN EFFICIENT ALGORITHM FOR SORTING A SET OF ELEMENTS WITH PARENT-CHILD RELATIONSHIPS

Pham Van Viet
Le Quy Don Technical University

ARTICLE INFO

Received: 21/7/2023
Revised: 30/8/2023
Published: 31/8/2023

KEYWORDS

Sorting algorithm
Elements with parent-child relationships
Algorithm analysis
AVL tree
Data structures and algorithms

ABSTRACT

A real world system may have a database table of records with parent-child relationships. The records may not be inserted into the table in family tree order. This paper proposes an efficient algorithm to solve the problem of sorting a list of elements with parent-child relationships that correspond to all the records in the database table. It is assumed that all the records are loaded into memory as a list of elements. The algorithm first creates a list *S* with the first element storing representatives of root elements and each other element storing a representative set of children of an element in the input list. The list *S* forms a set of trees of input elements. Then, each input element is added to the output sorted listed by performing a depth-first search on each tree. The algorithm takes *O(nlog(n))* time and *O(n)* space to solve the problem where *n* is the number of elements. The algorithm takes about 154 (seconds) to sort 2,441,405 elements using a laptop with an Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz 2.60 GHz processor, and 8 GB of RAM.

MỘT THUẬT TOÁN HIỆU QUẢ ĐỂ SẮP XẾP MỘT TẬP CÁC PHẦN TỬ VỚI QUAN HỆ CHA CON

Phạm Văn Việt
Đại học Kỹ thuật Lê Quý Đôn

THÔNG TIN BÀI BÁO

Ngày nhận bài: 21/7/2023
Ngày hoàn thiện: 30/8/2023
Ngày đăng: 31/8/2023

TỪ KHÓA

Thuật toán sắp xếp
Các phần tử với quan hệ cha-con
Phân tích thuật toán
Cây AVL
Các cấu trúc dữ liệu và thuật toán

TÓM TẮT

Một hệ thống trong thế giới thực có thể có một bảng cơ sở dữ liệu gồm các bản ghi có mối quan hệ cha-con. Các bản ghi có thể không được chèn vào bảng theo thứ tự cây gia đình. Bài báo này đề xuất một thuật toán hiệu quả để giải bài toán sắp xếp danh sách các phần tử có quan hệ cha-con tương ứng với tất cả các bản ghi trong bảng cơ sở dữ liệu. Giả sử rằng tất cả các bản ghi được tải vào bộ nhớ dưới dạng danh sách các phần tử. Đầu tiên, thuật toán tạo một danh sách *S* với phần tử đầu tiên lưu trữ các đại diện của các phần tử gốc và mỗi phần tử khác lưu trữ tập đại diện của các con của một phần tử trong danh sách đầu vào. Danh sách S tạo thành một tập các cây của các phần tử đầu vào. Sau đó, mỗi phần tử đầu vào được thêm vào danh sách được sắp xếp đầu ra bằng cách thực hiện tìm kiếm theo chiều sâu trên mỗi cây. Thuật toán cần thời gian *O(nlog(n))* và không gian *O(n)* để giải bài toán trong đó *n* là số phần tử. Thuật toán mất khoảng 154 (giây) để sắp xếp 2.441.405 phần tử, sử dụng máy tính xách tay có bộ xử lý Intel(R) Core(TM) i7-4510U CPU @ 2,00GHz 2,60 GHz và RAM 8 GB.

DOI: https://doi.org/10.34238/tnu-jst.8368

Email: v.v.pham2012@gmail.com

1. Introduction

A real world system may have a database table of records with parent-child relationships where a record has two identity fields: its own identity and its parent's identity for referring to its parent record. A record may not be inserted into the table in family tree order, in which an element appears after its parent, its older brothers, and its older brothers' descendants if they exist. This paper addresses the problem of sorting a list of elements with parent-child relationships that correspond to all the records in the database table. It is assumed that all the records are loaded into memory as a list of elements. An element of the list in memory corresponds to a record in the database table. A sample input list is given in Table 1. Its sorted list is given in Table 2. The pos, id, $parID$, $birthOrd$, and $assData$ columns in Table 1 are the position, identity, parent identity, birth order, and associated data of an element in the input list. Table 2 has all the columns of Table 1, and the $level$ column which is the level of an element (or the depth of an element). The table shows an element' id with a number of spaces before it, corresponding to the element's level. The greater the number of spaces, the higher the level. If an element has no parents, its $parID$ is $None$. The element is a root element. A root element also has $birthOrder$ as other elements.

Table 1. *Input list*

pos	id	parID	birthOrd	assData
0	0	None	1	Data 0
1	1	None	2	Data 1
2	2	0	3	Data 2
3	3	0	2	Data 3
4	4	1	2	Data 4
5	5	1	1	Data 5
6	6	0	1	Data 6
7	7	3	2	Data 7
8	8	3	1	Data 8

Table 2. *Output sorted list*

pos	id	parID	birthOrd	assData	level
0	0	None	1	Data 0	0
1	6	0	1	Data 6	1
2	3	0	2	Data 3	1
3	8	3	1	Data 8	2
4	7	3	2	Data 7	2
5	2	0	3	Data 2	1
6	1	None	2	Data 1	0
7	5	1	1	Data 5	1
8	4	1	2	Data 4	1

There have been a variety of studies on sorting a list of elements without parent-child relationships [1] – [5]. However, to the best of our knowledge, there hasn't been any study published on sorting elements with parent-child relationships. Sorting in general can be categorized into comparison sorting and integer sorting [3]. In comparison sorting, there are no limitations on an element's key field that is used for sorting (e.g., a real number field). A list of n-elements can be sorted by comparison based algorithms with a time complexity of $O(n^2)$ or $O(nlog(n))$ in the worst case. For example, insertion sort [6] and quicksort [7] run in $O(n^2)$ time; merge sort [8] and heapsort [9], [10] run in $O(nlog(n))$ time. The problem of sorting an n-element set which can be reduced to the problem of sorting a set of n-integers, can be solved by algorithms with better time complexity, but with some limitations on the sorted integers. For example, counting sort [11] has a time complexity of $O(n)$, where input integers are in the range 1 to k and k is $O(n)$; radix sort [12] runs in linear time if the number of digits in an input integer is constant and the range of a digit is from 1 to k where k is $O(n)$. In [3], Yijie Han proposed a method to convert real numbers to integer numbers in $O(n\sqrt{\log(n)})$ time, and then integer numbers are sorted with an integer sorting algorithm. In this paper, the problem of sorting a list of elements with parent-child relationships is classified as comparison sorting. We propose an efficient algorithm that takes $O(nlog(n))$ time and $O(n)$ space to solve the problem.

The following sections include: section 2 presents the proposed algorithm, section 3 presents the algorithm's analysis, and the last section is the conclusion.

2. Proposed algorithm

This section presents the algorithm for sorting a set of elements with parent-child relationships and shows how the algorithm works with the input elements given in Table 1.

The proposed algorithm is illustrated in Algorithm 1, in Python pseudocode. The algorithm first creates a list S with the first element storing representatives of root elements and each other element storing a representative set of children of an element in the input list L. The list S forms a set of trees of input elements. Then, each input element is inserted into the output sorted list sL by performing a depth-first search on each tree. The details of the algorithm are described below.

Lines 1-5 initialize a height-balanced search tree AVL tree: $avlTree4F$ (for more details about AVL tree, see [13], [14]). A node in $avlTree4F$ has two pieces of information: id (that is the identity of an element in the input list L) and pos (that is the position of the element in the input list L). The $avlTree4F$ stores all the ids and positions of the elements in the input list. It is used for efficiently finding the position of an element in the input list by its id.

In lines 6-9, a list S with a size of $n + 1$ is initialized. An element in S is an AVL tree. The element $S[0]$ is an AVL tree storing representatives of elements without parents in the input list. The element $S[i + 1]$ where $i \geq 0$ is an AVL tree storing representatives of children of the element $L[i]$. A representative element in an AVL tree's node corresponds to an element in the input list, and it has two pieces of information: id (that is the birth order of the coresponding element) and pos (that is the position of the coresponding element). The list S forms a set of trees of input elements.

In lines 10-19, each element in the input list L is visited. If the element $L[i]$ has no parents, then its representative element $(birthOrd, i)$ is inserted into $S[0]$. If the element has a parent, then the position (pos) of the parent is searched using the AVL tree $avlTree4F$. The element's representative $(birthOrd, i)$ is inserted into $S[pos + 1]$.

In lines 20-41, the output sorted list sL is constructed. Each input element is inserted into the output sorted list by performing a depth-first search on each tree in S using stack $pStack$ (a stack of positions of elements in S). First, in lines 23-25, if $S[0]$ is not empty, the position 0 (the position of the element $S[0]$ that contains representatives of root elements in the input list) is pushed into $pStack$. This paper assumes that any input element except for root elements is a descendant of a root element. Then, the while loop of lines 28-41 executes as long as $top \geq 0$ (or $pStack$ is not empty). Each iteration considers the top element $tPos$ in $pStack$. If $S[tPos]$ is empty, $tPos$ is popped out (line 32). If $S[tPos]$ is not empty, the representative element $item$ with minimum $birthOrd$ in $S[tPos]$ is popped out (line 34); the element in the input list at $item.pos$ is inserted into the sorted list sL (line 36), and the $level$ field equal to top is added to it (line 37). If $S[item.pos + 1]$ is not empty, the position $item.pos + 1$ is pushed into $pStack$. Here, $item.pos + 1$ is the position of $S[item.pos + 1]$ that contains representatives of children of $L[item.pos]$.

The rest of this section presents the steps of the algorithm for sorting the input list L of 9 elements in Table 1. First, the AVL tree $avlTree4F$ is constructed with 9 representatives of elements in the input list. A representative element has the structure (id, pos) that are the id and position of the element in the input list. The 9 representative elements are (0, 0), (1, 1), (2, 2),... and (8, 8). Then, S is initialized with 10 elements. An element in S is an AVL tree of representatives with the structure $(birthOrd, pos)$ containing the birth order and the position of an element in the input list L. $S[0]$ contains representatives of root elements in L: (1, 0), (2, 1). $S[1]$ includes representatives of children of the element at position 0 in L: (2, 3), (1, 6), (3, 2). S[2] contains representatives of children of the element at position 1 in L: (2, 4), (1, 5). $S[4]$ contains representatives of children of the element at position 3 in L: (2, 7), (1, 8). $S[3]$, $S[5]$, $S[6]$, $S[7]$, $S[8]$, and $S[9]$ are empty because the elements at positions 2, 4, 5, 6, 7, and 8 in L

have no children. $S[0]$ is not empty, so the stack $pStack$ is initialized with only one position of 0 (the position of $S[0]$). The value of top (the position at the top of pStack) is 0.

Algorithm 1. *Sorting algorithm*

```
    Input: An input list L of n-elements with parent-child relationships. An element has two identity
    fields: its own identity and its parent's identity, as presented in Table 1. The elements of L are
    numbered from 0.
    Output: An output sorted list sL of elements, in which an element appears after its parent, its
    older brothers, and its older brothers' descendants if they exist.
1   # Create the AVL tree avlTree4F, storing representatives of input elements in L, to efficiently find
    the position of an input element by its id.
2   avlTree4F = AVLTree()
3   for i in range(n):
4       # Insert a tree node that has id of L[i].id and pos of i (the position of L[i]).
5       avlTree4F.insert(L[i].id, i)
6   # Create the list S: S[0] stores representatives of input elements without parents, and S[i+1] with
    i ≥ 0 stores representatives of children of L[i].
7   S = []*(n+1)
8   for i in range(n+1):
9       S.append(AVLTree())
10  for i in range(n):
11      if (L[i].parID==None):
12          S[0].insert(L[i].birthOrd, i)
13      else:
14          # Find the tree node corresponding to the parent of L[i] in avlTree4F.
15          temp = avlTree4F.find(L[i].parID)
16          # The position of the parent of L[i] is pos. L[i] is a child of L[pos].
17          pos = temp.pos
18          # Insert a representative of L[i] into S[pos+1].
19          S[pos+1].insert(L[i].birthOrd, i)
20  top = -1
21  pStack = []
22  # Append the position of the representative set of elements having no parents.
23  if (S[0] is not empty):
24      top = top + 1
25      pStack.append(0)
26  count = 0
27  sL=[None]*n
28  while (top>=0):
29      tPos = pStack [top]
30      if (S[tPos])is empty):
31          top = top -1
32          pStack.pop()
33      else:
34          item= S[tPos].popMin()
35          count = count + 1
36          sL[count-1]=L[item.pos]
37          sL[count-1].level=top
38          if (S[item.pos+1] is not empty):
39              top = top+1
40              # Append the position of the representative set of children of L[item.pos] in S.
41              pStack.append(item.pos+1)
```

Next, the while loop of lines 28-41 constructs the output sorted list sL. The input for the while loop includes L, S, $pStack$, and top. The while loop outputs the sorted list sL with $count$ elements. Table 3 shows the steps of the while loop. It has two columns: **step** and the output of a

step, including: ***pStack***; ***top***; the list ***S*** whose element stores a set of representatives presented in the form of ***(birthOrd, pos)***; and the output sorted list ***sL*** of elements presented in the form of ***(id, parID, birthOrd, level).*** The initial step shows the input for the while loop. Other steps are described below. A variable in a step that changed from the previous step is presented in bold.

Step 1: The top element of $pStack$, $tPos$**,** is 0. The element in S at position $tPos$, $S[0]$, is considered. $S[0]$ is not empty, so the element $item$ with the smallest $birthOrd$ in $S[0]$ that is (1, 0) is popped out; the input element in L at position $item.pos$ 0, $L[0]$, is inserted into the output sorted list sL and then the $level$ field with the value of top, 0, is added to it. The representative set of children of the element in L at $item.pos$ 0, $S[1]$, is not empty, so top increases by 1 to 1 and the position of $S[1]$ is added to $pStack$ at the top of the stack. The stack $pStack$ is {0, 1}. $S[0]$ now contains only (2, 1). The sorted list sL with elements presented by structure $(id, parID, birthOrd, level)$ is {(0, $None$, 1, 0)}.

Step 2: The top element of $pStack$, $tPos$, is 1. The element in S at position $tPos$, $S[1]$, is considered. $S[1]$ is not empty, so the element $item$ with the smallest $birthOrd$ of $S[1]$ that is (1, 6) is popped out; the input element in L at position $item.pos$ 6 is inserted into the output sorted list sL and then the $level$ field with the value of top, 1, is added to it. The representative set of children of the element in L at $item.pos$ 6, $S[7]$, is empty, so the position of $S[7]$ is not added to $pStack$. The stack $pStack$ is unchanged. $S[1]$ after popping (1, 5) is now: {(2, 3), (3, 2)}. The sorted list sL with elements presented by structure (*id, parID, birthOrd, level*) is now: {(0, None, 1, 0), (6, 0, 1, 1)}.

Step 3: The top element of $pStack$, $tPos$, is 1. The element in S at position $tPos$, $S[1]$, is considered. $S[1]$ is not empty, so the element $item$ with the smallest $birthOrd$ of $S[1]$ that is (2, 3) is popped out; the input element in L at position $item.pos$ 3 is inserted into the output sorted list sL and then the $level$ field with the value of top, 1, is added to it. The representative set of children of the element at $item.pos$ 3, $S[4]$, is not empty, so the value of top increases by 1 to 2, the position of $S[4]$ is added to $pStack$ at the top of $pStack$. The stack $pStack$ is now {0, 1, 4}. $S[1]$ after popping (2, 3) is now: {(3, 2)}. The sorted list sL with elements presented by structure $(id, parID, birthOrd, level)$ is now: {(0, None, 1, 0), (6, 0, 1, 1), (3, 0, 2, 1)}.

Step 4: The top element of $pStack$, $tPos$, is 4. The element in S at position $tPos$ 4, $S[4]$, is considered. $S[4]$ is not empty, so the element with the smallest $birthOrd$ of $S[4]$ that is (1, 8) is popped out; the input element in L at position $item.pos$ 8 is inserted into the output sorted list sL and then the $level$ field with the value of top, 2, is added to it. The representative set of children of the element at $item.pos$ 8, $S[9]$, is empty, so the position of $S[9]$ is not added to $pStack$. The stack $pStack$ is unchanged. $S[4]$ after popping (1, 8) is now: {(2, 7)}. The sorted list sL with elements presented by structure $(id, parID, birthOrd, level)$ is now: {(0, $None$, 1, 0), (6, 0, 1, 1), (3, 0, 2, 1), (8, 3, 1, 2)}.

Step 5: The top element of $pStack$, $tPos$**,** is 4. The element in S at position $tPos$ 4, $S[4]$, is considered. $S[4]$ is not empty, so the element $item$ with the smallest $birthOrd$ of $S[4]$ that is (2, 7) is popped out; the input element at position $item.pos$ 7 is inserted into the output sorted list sL and then the $level$ field with the value of top, 2, is added to it. The representative set of the children of the element at $item.pos$ 7, $S[8]$, is empty, so the position of $S[8]$ is not added to $pStack$. The stack $pStack$ is unchanged. $S[4]$ after popping (1, 8) is now empty. The sorted list sL with elements presented by structure (*id, parID, birthOrd, level*) is now: {(0, $None$, 1, 0), (6, 0, 1, 1), (3, 0, 2, 1), (8, 3, 1, 2), (7, 3, 2, 2)}.

Step 6: The top element of $pStack$, $tPos$, is 4. The element in S at position $tPos$ 4, $S[4]$, is considered. $S[4]$ is empty, so the value of top decreases by 1 to 1; the top element of $pStack$, 4, is popped from the stack. The stack $pStack$ after popping 4 is {0, 1}. The sorted list sL is unchanged.

The next steps to step 12 operate similarly to the above steps.

Step 13: The top element of $pStack$, $tPos$, is 0. The element in S at position $tPos$ 0, $S[0]$, is considered. $S[0]$ is empty, so the value of top decreases by 1 to -1; the top element of $pStack$, 0, is popped from the stack. The stack $pStack$ after popping 0 is empty. The sorted list sL is unchanged. The while loop terminates.

Table 3. *The outputs of the algorithm*

Step	***pStack*; *top*; *S (birthOrd, pos)*; *sL (id, parID, birthOrd, level)***
Init	*pStack*:{0}; *top* = 0; *S*: {*S*[0]: (1, 0) (2, 1); *S*[1]: (2, 3) (1, 6) (3, 2); *S*[2]: (2, 4) (1, 5); *S*[4]: (2, 7) (1, 8)} *sL*: {}
1	***pStack*:** {0, **1**}; ***top* = 1**; ***S***: **{*S*[0]**: (2, 1); *S*[1]: (2, 3) (1, 6) (3, 2); *S*[2]: (2, 4) (1, 5); *S*[4]: (2, 7) (1, 8)}; ***sL*:** {**(0, None, 1, 0)}**
2	*pStack*: {0, 1}; *top* = 1; ***S***: {*S*[0]: (2, 1); ***S*[1]:** (2, 3) (3, 2)**;** *S*[2]: (2, 4) (1, 5); *S*[4]: (2, 7) (1, 8)}; ***sL***: {(0, None, 1, 0), **(6, 0, 1, 1)}**
3	***pStack***: {0, 1, **4**}; ***top* = 2**; ***S***: {*S*[0]: (2, 1); ***S*[1]**: (3, 2)**;** *S*[2]: (2, 4) (1, 5); *S*[4]: (2, 7) (1, 8)}; ***sL***: {(0, None, 1, 0) (6, 0, 1, 1) **(3, 0, 2, 1)}**
4	*pStack*: {0, 1, 4}; *top* = 2; ***S***: {*S*[0]: (2, 1); *S*[1]: (3, 2); *S*[2]: (2, 4) (1, 5); ***S*[4]**: (2, 7)}; ***sL***: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) **(8, 3, 1, 2)**}
5	*pStack*: {0, 1, 4}; *top* = 2; ***S***: {*S*[0]: (2, 1); *S*[1]: (3, 2); *S*[2]: (2, 4) (1, 5); ***S*[4]**: ∅**};** ***sL***: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) **(7, 3, 2, 2)}**
6	***pStack***: {0, 1}; ***top* = 1**; *S*: {*S*[0]: (2, 1); *S*[1]: (3, 2); *S*[2]: (2, 4) (1, 5)}; *sL*: {(0, None, 1, 0), (6, 0, 1, 1), (3, 0, 2, 2), (8, 3, 1, 2), (7, 3, 2, 2)}
7	*pStack*: {0, 1}; *top* = 1; ***S***: {*S*[0]: (2, 1); ***S*[1]**: ∅; *S*[2]: (2, 4) (1, 5)}; ***sL***: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) (7, 3, 2, 2) **(2, 0, 3, 1)**}
8	***pStack***: {0}; ***top* = 0**; *S*: {S[0]: (2,1); *S*[2]: (2, 4) (1, 5)}; *sL*: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) (7, 3, 2, 2) (2, 0, 3, 1)}
9	***pStack***: {0, **2**}; ***top* = 1**; ***S***: {***S*[0]**: ∅; *S*[2]: (2, 4) (1, 5);} ***sL***: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) (7, 3, 2, 2) (2, 0, 3, 1) **(1, None, 2, 0)}**
10	*pStack*: {0, 2}; *top* = 1; ***S***: {***S*[2]:** (2, 4)**}**; ***sL***: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) (7, 3, 2, 2) (2, 0, 3, 1) (1, None, 2, 0) **(5, 1, 1, 1)**}
11	*pStack*: {0, 2}; *top* = 1*;* ***S***: {***S*[2]**: ∅**}**; ***sL***: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) (7, 3, 2, 2) (2, 0, 3, 1) (1, None, 2, 0) (5, 1, 1, 1) **(4, 1, 2, 1)}**
12	***pStack***: {0}; ***top* = 0**; *S*: {} *sL*: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) (7, 3, 2, 2) (2, 0, 3, 1) (1, None, 2, 0) (5, 1, 1, 1) (4, 1, 2, 1)}
13	***pStack***: **{}**; ***top* = -1**; S: {} sL: {(0, None, 1, 0) (6, 0, 1, 1) (3, 0, 2, 1) (8, 3, 1, 2) (7, 3, 2, 2) (2, 0, 3, 1) (1, None, 2, 0) (5, 1, 1, 1) (4, 1, 2, 1)}

3. Algorithm analysis

This section presents the proposed algorithm's time and memory complexities.

The algorithm first initializes the AVL tree $avlTree4F$ in line 2. This operation takes time $O(1)$. The for loop of lines 3-5 has n iterations. At iteration i, insert operation is executed on the AVL tree with i nodes (i from 0 to $n-1$). The time complexity of the insert operation in the worst case is $O(\log(n))$ where the number of nodes in the AVL tree is $O(n)$ [14]. Therefore, the for loop of lines 3-5 takes time $O(nlog(n))$. The initialization of list S with a size of $n+1$ in line 7 takes time $O(n)$. The for loop of lines 8-9 takes time $O(n)$, since the time complexity of AVL tree initialization is $O(1)$. The for loop of lines 10-19 has n iterations. At iteration i, if the element $L[i]$ doesn't have a parent, an insert operation on the AVL tree $S[0]$ is executed.

Otherwise a find operation on $avlTree4F$ and an insert operation on an AVL tree element in the list S are executed. The maximum size of $avlTree4F$ and an AVL tree in S is n, so the time complexity of the find and insert operations is $O(nlog(n))$ [14]. Thus, the time complexity of lines 10-19 is $O(nlog(n))$. Lines 20-26 take time $O(1)$. Line 27 takes time $O(n)$ to initialize list sL with a size of n. The while loop of lines 28-41 takes time $O(nlog(n))$. At an iteration of the while loop, at most one of the two operations of pushing and popping on $pStack$ is executed. Since the size of S is $n+1$, the number of positions of elements in S pushed into $pStack$ is at most $n+1$, and the number of positions of elements in S popped from it is at most the same. Hence, the while loop has at most $2(n+1)$ iterations. The time complexity of each line from 29 to 41 is $O(1)$, except for line 34. Line 34 pops the minimum element from an AVL tree with a size of at most n. The pop operation first finds the minimum element in the AVL tree by going from the root to the leftmost node. This takes $O(\log(n))$ time, since the height of the AVL tree is $O(\log(n))$ [10]. Then the minimum element is deleted from the tree. This takes $O(\log(n))$ time [14]. Hence, line 34 takes $O(\log(n))$ time. Therefore, the time complexity of lines 28-41 is $O(nlog(n))$. The overall time of the algorithm is $O(nlog(n))$.

The algorithm ran with programmatically generated sets of elements with parent-child relationships to different levels. At level 0, five elements without parents were generated. Then, five children of each element at a level were generated. After a set of n elements to some level was generated, n couples of positions were generated randomly, and n couples of elements corresponding to the n couples of positions were swapped. Table 4 shows the running times using a laptop with an Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz 2.60 GHz processor, and 8 GB of RAM for sets of elements generated to levels 2 to 8. The number of elements in a set generated to level m is $\sum_{i=1}^{m+1} 5^i = (5^{m+2}-5)/4$. The number of elements in a set generated to a level is about five times higher than that in a set generated to the previous level. The algorithm ran 10 times for each generated set of elements. Average running times (in ms) are reported in the table. The running time for a set generated to a level is about 5 to 6 times higher than that for the set generated to the previous level. The algorithm takes about 154 (s) to sort 2,441,405 elements.

Table 4. *Running time*

Level	#Elements	Time (ms)
2	155	4.197
3	780	23.583
4	3,905	131.735
5	19,530	714.290
6	97,655	4276.553
7	488,280	26330.328
8	2,441,405	154513.434

If the input list L and output list sL with a size of n are not taken into account, the algorithm uses the two lists: $pStack$ and S; the AVL tree $avlTree4F$; and a number of variables with a memory space of $O(1)$. The maximum size of $pStack$ doesn't exceed the highest level of an element plus one. The list $pStack$ has a maximum size of $n+1$ in the worst case when each input element has only one child. An element in $pStack$ stores the position value of an element in S, which has memory space $O(1)$, so $pStack$ has memory space $O(n)$. S is a list of AVL trees. $S[0]$ stores representatives of elements without parents in the input list L. $S[i+1]$ where $i \geq 0$ stores representatives of children of the element $L[i]$. A node in an AVL tree in S has two representative fields: $birthOrd$ (used as the key of the node corresponding to the representative) and position of an element in the input list, and four other fields to form an AVL tree, including the node's height and three references to its left, right, and parent nodes. An element in the input list has only one parent, so it has only one representative in S. Hence, the total number of representative elements stored in S is n. The AVL tree $avlTree4F$ for finding the position of an

element in the input list by its id has n elements. A node in $avlTree4F$ has six fields as in an AVL tree in S, except for the key field being an input element's id. Therefore, overall the memory space of the algorithm is $O(n)$.

4. Conclusion

This paper proposed an efficient algorithm using the AVL tree data structure to sort a set of n-elements with parent-child relationships in time $O(nlog(n))$ and memory space $O(n)$. The algorithm can sort 2,441,405 elements in about 154 (seconds) using a laptop with an Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz 2.60 GHz processor, and 8 GB of RAM. To the best of our knowledge, the paper is the first published study about sorting a set of elements with parent-child relationships. We hope that the work in this paper is the basis for solving the problem with more efficient data structures and algorithms.

REFERENCES

[1] D. Knuth, "Chapter 5 - Sorting," in *The Art of Computer Programming*, 3rd ed., vol. 3: Sorting and Searching, Addison-Wesley, 1997, pp. 1-391.

[2] Y. Han, "Deterministic sorting in O(nloglogn) time and linear space," *Journal of Algorithms*, vol. 50, no. 1, pp. 96-105, 2004.

[3] Y. Han, "Sort Real Numbers in $O(n\sqrt{\log(n)})$ Time and and Linear Space," *Algorithmica*, vol. 82, no. 2, pp. 966-978, 2020.

[4] S. Abdel-Hafeez and A. Gordon-Ross, "An Efficient O(N) Comparison-Free Sorting," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 6, pp. 1930-1942, 2017.

[5] F. C. Leu, Y. T. Tsai, and C. V. Tang, "An efficient external sorting algorithm," *Information Processing Letters*, vol. 75, no. 4, pp. 159-163, 2000.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Section 2.1 Insertion sort," in *Introduction to Algorithms*, 4th ed., The MIT Press, 2022, pp. 17-33.

[7] R. Sedgewick, "Implementing Quicksort programs," *Communications of the ACM*, vol. 21, no. 10, pp. 847-857, 1978.

[8] D. Knuth, "Sorting by Merging," in *The Art of Computer Programming*, 3rd ed., vol. 3: Sorting and Searching, Addison-Wesley, 1998, pp. 158-166.

[9] J. W. J. Williams, "Algorithm 232 (HEAPSORT)," *Communications of the ACM*, vol. 7, no. 6, pp. 347-348, 1964.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Section 6 Heap sort," in *Introduction to Algorithms*, 4th ed., The MIT Press, 2022, pp. 161-172.

[11] H. H. Seward, "2.4.6 Internal Sorting by Floating Digital Sort," in *Information sorting in the application of electronic digital computers to business operations*, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954, pp. 25-28.

[12] D. E. Knuth., "Section 5.2.5: Sorting by Distribution," in *The Art of Computer Programming*, 3rd ed., vol. 3: Sorting and Searching, Addison-Wesley, 1997, pp. 168-179.

[13] G. Adelson-Velskii and E. Landis, "An algorithm for the organization of information," in *the USSR Academy of Sciences (in Russian)*, 1962.

[14] P. Brass, "3.1 Height-Balanced Trees," in *Advanced Data Structures*, Cambridge University Press, 2008, pp. 50-61.